In [1]:
```python
from pathlib import Path
from typing import List

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import matplotlib; matplotlib.use("tkAgg")
import matplotlib.pyplot as plt
from sklearn.metrics import (
    confusion_matrix,
    ConfusionMatrixDisplay,
    roc_curve,
    auc,
    RocCurveDisplay,
    precision_recall_curve,
    PrecisionRecallDisplay,
)
from sklearn.preprocessing import StandardScaler
```

In [2]:
```python
# A simple Transformer-based classifier for sequence data
class TransformerClassifier(nn.Module):

    def __init__(
        self,
        input_size: int = 1,      # Number of features per time step
        seq_length: int = 12,     # Length of input sequences
        d_model: int = 64,        # Size of embedding / hidden dimension
        nhead: int = 4,           # Number of attention heads
        num_layers: int = 2,      # Number of Transformer layers
        dropout: float = 0.3,     # Dropout rate for regularisation
    ):
        super().__init__()

        # Project input features to model dimension
        self.input_projection = nn.Linear(input_size, d_model)

        # One encoder layer: self-attention + feed-forward
        enc_layer = nn.TransformerEncoderLayer(
            d_model=d_model, nhead=nhead, dropout=dropout, batch_first=True
        )

        # Stack multiple encoder layers
        self.encoder = nn.TransformerEncoder(enc_layer, num_layers=num_layers)

        # Final classification head: transform to a single output
        self.classifier = nn.Sequential(
            nn.Linear(d_model, 32),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(32, 1)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # project input to model dimension
        x = self.input_projection(x)

        # apply Transformer encoder
        x = self.encoder(x)

        # average across sequence (global mean pooling)
        pooled = x.mean(dim=1)

        # classify and apply sigmoid (output between 0 and 1)
        return torch.sigmoid(self.classifier(pooled)).squeeze()
```

In [3]:

```
# Wrap features (X) and labels (y) into a DataLoader for batching
def _to_loader(X, y, batch_size: int, shuffle: bool = False):
    dataset = list(zip(X, y))  # Combine inputs and labels
    return torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=shuf


# Train the model for one epoch (one full pass through the data)
def train_one_epoch(model, loader, criterion, optim):
    model.train()  # Set model to training mode
    running = 0.0  # Track total loss

    for xb, yb in loader:  # Loop through batches
        optim.zero_grad()  # Reset gradients
        loss = criterion(model(xb), yb)  # Compute loss
        loss.backward()  # Backpropagate
        optim.step()  # Update weights
        running += loss.item() * xb.size(0)  # Accumulate batch loss

    return running / len(loader.dataset)  # Return average loss


# Evaluate the model's accuracy
def evaluate(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0

    with torch.no_grad():  # No gradient tracking needed
        for xb, yb in loader:
            preds = (model(xb) >= 0.5).float()  # Predict and round to 0 or 1
            correct += (preds == yb).sum().item()  # Count correct predictions

    return correct / len(loader.dataset)  # Return accuracy
```

In [4]:

```
# Manually performs cross-validation using custom folds in your DataFrame
def cross_validate_manual(
    df: pd.DataFrame,
    feature_cols: List[str],     # List of columns to use as features
    epochs: int = 20,
    batch_size: int = 64,
    lr: float = 3e-4,            # Learning rate
    weight_decay: float = 1e-3,  # Weight decay for regularization
):
    results = []  # Store best validation accuracy for each fold
```

```python
    # Loop through each unique fold number
    for fold in sorted(df["Fold"].unique()):
        print(f"\n── Fold {fold + 1} / {df['Fold'].nunique()} ────────────────")

        # Split into training and validation sets
        train_df = df[df["Fold"] != fold]
        val_df = df[df["Fold"] == fold]

        # Normalize features and reshape for model input
        scaler = StandardScaler()
        X_train = scaler.fit_transform(train_df[feature_cols]).reshape(-1, 12, 1)
        X_val = scaler.transform(val_df[feature_cols]).reshape(-1, 12, 1)

        # Convert to PyTorch tensors
        y_train = train_df["Label"].values.astype(np.float32)
        y_val = val_df["Label"].values.astype(np.float32)
        X_train_t = torch.tensor(X_train, dtype=torch.float32)
        y_train_t = torch.tensor(y_train)
        X_val_t = torch.tensor(X_val, dtype=torch.float32)
        y_val_t = torch.tensor(y_val)

        # Create DataLoaders
        train_loader = _to_loader(X_train_t, y_train_t, batch_size, shuffle=True)
        val_loader = _to_loader(X_val_t, y_val_t, batch_size)

        # Initialize model, loss, and optimizer
        model = TransformerClassifier()
        criterion = nn.BCELoss()  # Binary classification loss
        optim = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_de

        best_acc = 0.0
        bad_epochs = 0
        patience = 5  # Early stopping if no improvement for 'patience' epochs

        # Training loop
        for epoch in range(1, epochs + 1):
            loss = train_one_epoch(model, train_loader, criterion, optim)
            val_acc = evaluate(model, val_loader)
            print(f"Epoch {epoch:02}/{epochs} – loss: {loss:.4f} – val acc: {val_ac

            # Save best model based on validation accuracy
            if val_acc > best_acc:
                best_acc, bad_epochs = val_acc, 0
                best_state = model.state_dict()
            else:
                bad_epochs += 1
                if bad_epochs == patience:
                    print("Early stopping")
                    break

        # Save best accuracy for the fold
        results.append(best_acc)

        # Save best model checkpoint
        Path("checkpoints").mkdir(exist_ok=True)
        torch.save(best_state, f"checkpoints/fold_{fold}.pt")
        print(f"Best acc fold {fold + 1}: {best_acc:.4f}")

    # Summary of results
    print("\n====== Validation Accuracy Summary ======")
    for i, acc in enumerate(results, 1):
        print(f"Fold {i}: {acc:.4f}")
    print(f"Mean Accuracy: {np.mean(results):.4f}")
    print(f"Standard Deviation: {np.std(results):.4f}")

    return results
```

In [5]:
```python
# Trains on the full dataset and shows evaluation visualisations
def visual_evaluation_full(df: pd.DataFrame, feature_cols: List[str]):
    # Scale features and reshape for the model
    scaler = StandardScaler()
    X = scaler.fit_transform(df[feature_cols]).reshape(-1, 12, 1)
    y = df["Label"].values.astype(np.float32)

    # Convert to PyTorch tensors
    X_t = torch.tensor(X, dtype=torch.float32)
    y_t = torch.tensor(y)

    # Create DataLoader
    loader = _to_loader(X_t, y_t, batch_size=64, shuffle=True)

    # Initialize model, loss function, and optimizer
    model = TransformerClassifier()
    criterion = nn.BCELoss()
    optim = torch.optim.AdamW(model.parameters(), lr=3e-4, weight_decay=1e-3)

    # Train for a fixed number of epochs
    for _ in range(15):
        train_one_epoch(model, loader, criterion, optim)

    # Get model predictions
    model.eval()
    with torch.no_grad():
        y_scores = model(X_t).cpu().numpy().ravel()  # Raw output scores
    y_pred = (y_scores > 0.5).astype(int)  # Convert scores to binary predictions

    # Confusion Matrix
    cm = confusion_matrix(y, y_pred)
    ConfusionMatrixDisplay(confusion_matrix=cm).plot(cmap="Blues")
    plt.title("Confusion Matrix")
    plt.savefig("confusion_matrix.png", dpi=300, bbox_inches="tight")
    plt.show(block=True)

    # ROC Curve
    fpr, tpr, _ = roc_curve(y, y_scores)
    roc_auc = auc(fpr, tpr)
    RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=roc_auc).plot()
    plt.title(f"ROC Curve (AUC = {roc_auc:.4f})")
    plt.savefig("roc_curve.png", dpi=300, bbox_inches="tight")
    plt.show(block=True)
    # Precision-Recall Curve
    precision, recall, _ = precision_recall_curve(y, y_scores)
    PrecisionRecallDisplay(precision=precision, recall=recall).plot()
    plt.title("Precision-Recall Curve")
    plt.savefig("pr_curve.png", dpi=300, bbox_inches="tight")
    plt.show(block=True)
```

In [6]:

```python
import time
import psutil
import os

if __name__ == "__main__":
    process = psutil.Process(os.getpid())

    # === Resource Monitoring Start ===
    overall_start_time = time.time()
    overall_start_ram = process.memory_info().rss / 1024 / 1024  # in MB
    overall_start_cpu = psutil.cpu_percent(interval=1)

    # Load dataset
    df = pd.read_csv("D:\\Coding Projects\\Detection-of-SYN-Flood-Attacks-Using-Mac
    feature_cols = df.columns.difference(["Label", "Fold"]).tolist()[:12]

    # Run cross-validation on Transformer
    results = cross_validate_manual(df, feature_cols)

    # === Resource Monitoring End ===
    overall_end_time = time.time()
    overall_end_ram = process.memory_info().rss / 1024 / 1024  # in MB
    overall_end_cpu = psutil.cpu_percent(interval=1)

    # === Summary ===
    print("\n Overall Training Stats ")
    print(f"Total Training Time: {overall_end_time - overall_start_time:.2f} second
    print(f"Total RAM Usage Increase: {overall_end_ram - overall_start_ram:.2f} MB"
    print(f"CPU Usage (at final check): {overall_end_cpu}%")
```

```
── Fold 1 / 5 ────────────────
Epoch 01/20 – loss: 0.2557 – val acc: 0.9969
Epoch 02/20 – loss: 0.0478 – val acc: 0.9969
Epoch 03/20 – loss: 0.0388 – val acc: 0.9969
Epoch 04/20 – loss: 0.0360 – val acc: 0.9969
Epoch 05/20 – loss: 0.0300 – val acc: 0.9969
Epoch 06/20 – loss: 0.0302 – val acc: 0.9964
Early stopping
Best acc fold 1: 0.9969

── Fold 2 / 5 ────────────────
Epoch 01/20 – loss: 0.2690 – val acc: 0.9958
Epoch 02/20 – loss: 0.0498 – val acc: 0.9958
Epoch 03/20 – loss: 0.0324 – val acc: 0.9958
Epoch 04/20 – loss: 0.0262 – val acc: 0.9958
Epoch 05/20 – loss: 0.0227 – val acc: 0.9958
Epoch 06/20 – loss: 0.0244 – val acc: 0.9870
Early stopping
Best acc fold 2: 0.9958

── Fold 3 / 5 ────────────────
Epoch 01/20 – loss: 0.1876 – val acc: 0.9927
Epoch 02/20 – loss: 0.0420 – val acc: 0.9927
Epoch 03/20 – loss: 0.0336 – val acc: 0.9927
Epoch 04/20 – loss: 0.0317 – val acc: 0.9927
Epoch 05/20 – loss: 0.0256 – val acc: 0.9927
Epoch 06/20 – loss: 0.0254 – val acc: 0.9927
Early stopping
Best acc fold 3: 0.9927

── Fold 4 / 5 ────────────────
Epoch 01/20 – loss: 0.2297 – val acc: 0.9932
Epoch 02/20 – loss: 0.0332 – val acc: 0.9958
Epoch 03/20 – loss: 0.0222 – val acc: 0.9964
Epoch 04/20 – loss: 0.0227 – val acc: 0.9917
Epoch 05/20 – loss: 0.0214 – val acc: 0.9927
Epoch 06/20 – loss: 0.0160 – val acc: 0.9964
Epoch 07/20 – loss: 0.0163 – val acc: 0.9969
Epoch 08/20 – loss: 0.0134 – val acc: 0.9969
Epoch 09/20 – loss: 0.0145 – val acc: 0.9969
Epoch 10/20 – loss: 0.0154 – val acc: 0.9969
Epoch 11/20 – loss: 0.0134 – val acc: 0.9969
Epoch 12/20 – loss: 0.0120 – val acc: 0.9969
Early stopping
Best acc fold 4: 0.9969

── Fold 5 / 5 ────────────────
Epoch 01/20 – loss: 0.2440 – val acc: 0.9932
Epoch 02/20 – loss: 0.0389 – val acc: 0.9932
Epoch 03/20 – loss: 0.0280 – val acc: 0.9932
Epoch 04/20 – loss: 0.0273 – val acc: 0.9917
Epoch 05/20 – loss: 0.0240 – val acc: 0.9932
Epoch 06/20 – loss: 0.0190 – val acc: 0.9932
Early stopping
Best acc fold 5: 0.9932
```

```
══════ Validation Accuracy Summary ══════
Fold 1: 0.9969
Fold 2: 0.9958
Fold 3: 0.9927
Fold 4: 0.9969
Fold 5: 0.9932
Mean Accuracy: 0.9951
Standard Deviation: 0.0018

 Overall Training Stats
Total Training Time: 153.87 seconds
Total RAM Usage Increase: 187.30 MB
CPU Usage (at final check): 7.7%
```